Selecting Neural Network Topologies: A Hybrid Approach Combining Genetic Algorithms and Neural Networks

By

Christopher M. Taylor

B.S. - Computer Science
Southwest Missouri State University, 1997

Submitted to the Department of Electrical Engineering and Computer Science and the Faculty of the Graduate School of the University of Kansas in partial fulfillment of the requirements for the degree of Master of Science

_____
Dr. Arvin Agah
(Committee Chair)

_____
Dr. Nancy Kinnersley
(Committee Member)

_____
Dr. John Gauch
(Committee Member)

_____
Date of Acceptance

Abstract

This work examines the use of genetic algorithms and neural networks to generate neural network topologies. The data set consists of digital images of objects taken from different angles. A successful neural network topology had been trained on this data, so it was investigated whether the genetic algorithm could evolve a neural network topology capable of learning the training data. The genetic algorithm is used to evolve populations of neural network topologies. The neural network is trained using each of the topologies, and the remaining error in training is used to provide a fitness value for each of the topologies. Thus, the fitness function is the neural network itself.

Acknowledgements

This work would not have been possible without the support, encouragement, and patience of several people who deserve recognition for their contributions.

The first person I would like to thank is, of course, my wife Kim Taylor. She has sacrificed more for me than I can ever repay, leaving behind her family and friends in the city where she had spent her entire life and following me in pursuit of my dream. I quit a great job to be a full-time student, and she has valiantly supported us on her salary. I'll gladly spend the rest of my life paying it all back to her.

Next, I would like to thank Dr. Arvin Agah, my thesis chairperson and academic advisor, for his tremendous patience and remarkable guidance. He has helped me keep focus, and has encouraged me through the difficulties of research and discovering that the results are very rarely what we would hope them to be.

Several professors have been instrumental in my success and helping me adjust from corporate life to academic research. Just by giving me a job, Dr. Nancy Kinnersley helped me tremendously and I'll be forever grateful. Thank you to Dr. John Gauch for helping me get my feet under me, and thank you Dr. Susan Gauch for giving me a second chance.

Lastly, I thank all my wonderful friends and family for believing in me. A special thank you goes to my father, Mike Taylor, who taught me that I can do anything if I only try.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

While artificial neural networks are typically robust enough that many different topologies can be used to learn the same set of data, the topology chosen still impacts the amount of time required to learn the data and the accuracy of the network in classifying new data. Thus, choosing a good topology becomes a crucial task to the success of the neural network.

The work that follows is an indirect result of another study. In the prior study, two neural networks were being used to investigate the "dual modality effect" in learning. The theory behind dual modality is that we rely on many different forms of memory and learning. For example, if one were to recall watching a movie, one would probably recall some scenes entirely while other scenes might be recalled only by image or only by sound. Sight and sound are the two most common modalities. By combining multiple modalities in learning, more data is encoded with the memory, thus making it easier to recall and process. For example, when listening to a lecture, if one's mind were to wander, then that portion of the lecture is missed completely and cannot be recalled. However, if there is a visual element to the lecture, then perhaps the image can be recalled even if the words spoken could not.

The previous study hoped to test this concept using two neural networks: one for sight and one for sound. Each network was to be trained on a set of four objects (toys capable of making sounds). Each object made a different sound. Digital pictures were taken of each object from different angles, and sounds were recorded

from each object. Only some of the angles would be used for training, and only some of the sounds would be trained in the other network. Once both networks were trained, they would be combined by a third network which would bring the pieces of data together to be able to recognize the objects by sight, by sound, and by both. The real test would be in determining how well the system could combine sight and sound to recognize an object from an angle not seen before or with a sound not heard before (i.e., captured from that orientation). In this manner, the dual modality concept could be tested in computing systems, indicating that perhaps data should be encoded in memory in multiple manners.

The training of the image network was successful on the first try. Unfortunately, the sound network did not train on the first attempt. Nor on the second attempt. Several different network topologies were selected and each failed to learn the sound data. A great deal of time was invested in trying to identify a structure that could be successfully trained with the sound data. Over twenty different network topologies were tested, requiring significant amounts of time. All twenty failed, and so that direction of the experiment was abandoned.

The study which follows is the result of the failures and frustrations from the previous experiment. It should be noted that the failure of the first experiment could be a result of the nature of the problem and associated data, i.e., it is possible that no network topology is capable of learning from the specific set of data used for the first experiment. Perhaps a network topology could be programmatically developed in such a way as to guarantee the success of the network, assuming that a topology

capable of learning the data exists. A review of the literature showed that genetic algorithms had been used in other studies to do this very thing.

Chapter 2

Background and Related Work

This thesis combines neural networks with genetic algorithms in an effort to find a better method of selecting neural network topologies. This chapter provides background information on neural networks, genetic algorithms, and hybrid systems like the one used here.

2.1 Neural Networks

Artificial Neural Networks (ANNs), are a method of artificial intelligence based upon models posed by cognitive theory in psychology as to how biological brains function. A brain is composed of neurons, cells that receive a stimulus which triggers a response from the neuron. The response from the neuron can trigger other neurons, which trigger other neurons, etc. Eventually, this chain of neural activation results in some response from the body, such as recollection of a memory, identifying a sound, movement of a muscle, etc.

2.1.1 The Structure of Neural Networks

The way ANNs are represented in a computer system is very similar. An artificial neural network consists of several neurons, but they are divided into layers. There is an “input” layer where the initial “stimulus” is received. These neurons are connected to a layer of “hidden” neurons. This hidden layer can be connected to either another hidden layer, or the “output” layer. There can be any number of hidden layers between the input layer and the output layer, but typically the number of

hidden layers in any particular ANN is limited. Typically, a neuron of any layer is connected to each other neuron in adjoining layers. Thus, each neuron in the input layer is connected to each neuron in the first hidden layer. Each neuron of the first hidden layer is also connected to each neuron in the next layer (either another hidden

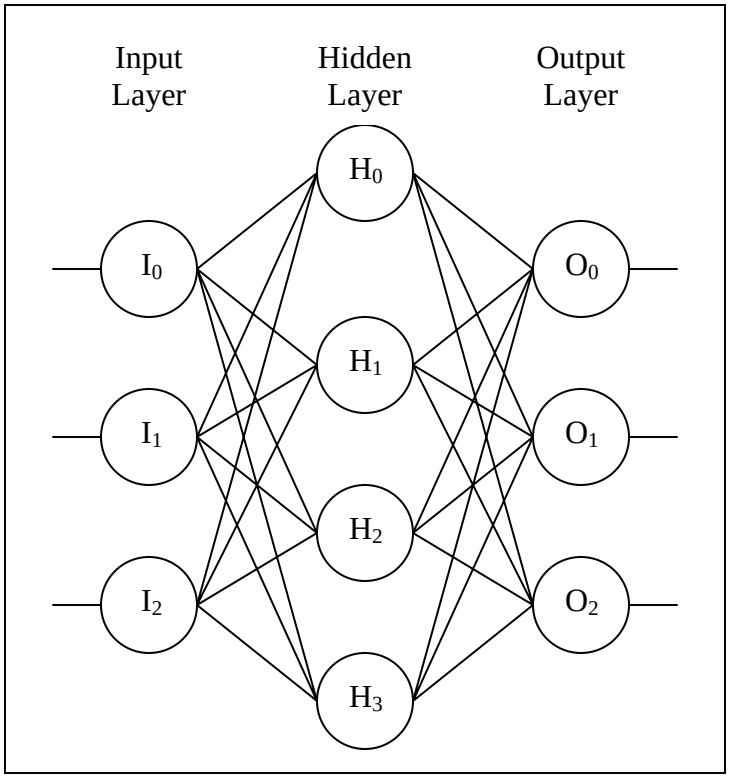

Figure 2.1. A neural network with 3 input neurons, one hidden layer with 4 hidden neurons, and 3 output neurons.

layer, or the output layer). Each of these connections between neurons has a "weight" associated with it, and each connection is treated individually – each connection can have a unique weight associated with it. As the network "learns," these weights are adjusted to represent the strength of connections between neurons. Thus, the network might determine that a particular connection is of no value, and give it a weight of zero. To determine the output from any particular neuron, the neuron must first

gather its input. For neurons in the input layer, this is trivial – the input is merely the value given as input to the network. For each other neuron, however, this is more complicated. The input for neurons not in the input layer is a function of the output from each neuron in the previous layer, and the weight of the connection to that neuron. Using Figure 2.1, the input to neuron H_0 in the hidden layer is based upon the output of each neuron in the input layer, multiplied by the connection between that neuron and H_0. To be more specific, suppose the output from neuron I_2 is 0.35 and the weight of 0.11 exists for the connection between I_2 and H_0. Multiplying 0.35 x 0.11, results in an input of 0.0385 into H_0 from I_2. To get the total input to H_0, the input values from each connected neuron must be summed. Thus, the input to a neuron j can be written as:

$$\sum(O_i * W_{i,j}) \qquad (1)$$

where O_i is the output from neuron i and $W_{i,j}$ is the weight of the connection between neuron i and neuron j. The total input to the neuron is then used to determine an output value from the neuron. A sigmoid function is typically used for this, and the output value from a neuron will be between 0 and 1. When a neuron is "inactive" its output will be close to 0. When a neuron is triggered or "active" then its value will be close to 1.

2.1.2 Training a Neural Network

In order to get a Neural Network to successfully learn a task, it must first be trained. This requires that some training data be gathered. The training data must

include the inputs to each input neuron, and the anticipated result from each output neuron. Thus, for Figure 2.1, each example in the training data must include 6 values: 3 for the inputs, and 3 for the outputs. Any number of examples can be provided to the network. The more training examples, the better the network can learn the data. There is a trade-off, however, between having a very accurate network and one that is robust to new data. By providing large amounts of training data, a network can train to be very accurate in its results, but will not be very successful at dealing with new examples not included in training. On the other hand, by providing fewer training examples, the network might be able to better handle new examples that it has never seen before, but will likely suffer in the overall accuracy of its results.

Once the training data has been obtained, the training process is relatively simple. When the network is first created, these weights often have random or pre-determined values and are unlikely to produce the expected outputs on the first attempt. Training or "learning" in an ANN takes place by changing the connection weights between neurons. The most common learning technique employed in neural networks is called "back propagation." Back propagation works by taking the input values and pulsing the network to see what outputs are obtained based on the current weights. The resulting outputs are then compared with the expected outputs indicated in the training data. If the outputs are different then the weights are adjusted based upon the amount of error involved and the "learning rate," λ. The learning rate is typically a value between 0.3 and 0.7 and represents the percentage of the error to be

used for adjusting the weights. A larger value for the learning rate means that the weights receive greater adjustments, while a smaller value indicates that the weights receive smaller adjustments. The learning rate is important because the neural network is supposed to learn all of the examples in the training data and over-adjusting the weights after each training example is used can cause the network to "unlearn" some of the progress made with earlier training examples. Thus, by reducing the λ the network can continue to learn new training examples without counter acting prior progress. As the λ-value is decreased, so does the speed at which the network learns. Setting a larger learning rate means the network will train faster, but might reach a limit at which it cannot improve because changes in weight values are interfering with learning.

Weights in a neural network are typically readjusted backwards – the weights between the output layer and the adjacent hidden layer are adjusted first and the weights between the input layer and the hidden layer connected to it are adjusted last. Thus, the error is propagated backwards. Once the weights have been re-adjusted, the next example in the training data is used and the weights again re-adjusted. Once every example in the training data has been used to re-adjust the weights, the process starts over again at the beginning, with the network again trying the first example in the training data and seeing how close the outputs match to those it was supposed to obtain. Each iteration through the training examples is called an "epoch." The network continues to train in this manner until a) the difference between the outputs from the network and the expected outputs, i.e., the error, falls below a certain

threshold (indicating that the network has successfully learned the data) or b) a maximum number of epochs have passed (indicating that the network failed to learn the data in the number of epochs it was allowed).

2.1.3 Applications of Neural Networks

Neural networks are most typically used in "classification" problems. A classification problem is one where a certain number of "classes" are identified. Based on the input data, the ANN determines to which class the sample will most likely belong. An example of a classification problem might be in determining if a picture is of a male or a female – a two class problem. When using a neural network for classification, the number of output nodes are usually determined by the number of classes, with one output node per class. In this manner, the values of each output node represent the likelihood of the sample falling into each of those classes, as determined by the neural network. It should be noted, however, that this is not probability – the ANN might indicate with absolute certainty that a given sample belongs to multiple classes, or none at all. For example, using the male/female classes mentioned earlier, the ANN might indicate that a particular photo is both male and female, or neither. This would indicate that the network requires further refinement in order to successfully classify the photo.

Neural networks have been used successfully in a large number of applications. They have been used to successfully recognize handwritten (Lyon, 1996) or spoken (Hosom, 1999) words, allowing for dictation software and software which can take handwritten text and convert it to electronic text. They have been

used to distinguish between the handwriting of different individuals, providing a method for handwriting verification (Guyon). ANN's have been used in face recognition, allowing the network to distinguish between pictures of different people. One such application is a crime fighting system used in Tampa (GAITS). Cameras take pictures of people on the streets and compare them to a database of photos to help police locate felons. With any classification problem, there is a potential for the application of an artificial neural network.

More information on artificial neural networks, see Bart Kosko's book "Neural Networks and Fuzzy Systems" (1996).

2.2 Genetic Algorithms

Genetic algorithms (GAs) are based upon evolutionary principles of natural selection, mutation, and survival of the fittest (Dulay). GAs are very different from most computer programs, which have well-defined algorithms for coming up with solutions to problems. The genetic algorithm approach is to generate a large number of potential solutions and "evolve" a solution to the problem.

2.2.1 How Genetic Algorithms Work

One of the big keys to a successful genetic algorithm is in the development of a good "fitness function." The fitness function is how each potential solution is evaluated by the algorithm, and is in essence how the problem to be solved by the algorithm is defined. For example, if the purpose of the genetic algorithm is to design a car, then the fitness function will provide a means for evaluating the "fitness" of any car design.

When developing a genetic algorithm, one must decide how each solution will be represented in the algorithm. For simplicity, a string of bits is most often used. The bits can be used to represent any part of the solution. Taking the car example mentioned above, some of the bits might represent the color of the car, others the size of the wheels, and others the gas mileage of the car. It is the responsibility of the fitness function to understand what the bits mean and how to use them to evaluate the fitness of each potential solution.

When the GA is first run, it generates an initial "population" of potential solutions, which could be random. Each member of the population is then examined and its fitness is evaluated and recorded. Once each member of the population has been evaluated, then the next generation is produced from the current generation. There are many ways of generating the next generation, but the two most popular techniques involve "crossover" and "mutation." In crossover, two members of the population are chosen at random with higher probability given to the more fit members of the population. These two members are then combined to produce two offspring. This is usually performed by selecting a position in the bit sequence and exchanging the two sequences after that position, which is called the crossover point.

For example, given the following two members of a population:

11101010

10010100

If these were crossed over at position 4, the resulting offspring would be:

11100100

10011010

There are many ways to perform the crossover, but this example is one of the simplest. The result is two new members in the next generation. In theory, these two new members should be reasonably better fit because they likely came from fit members in the previous population. Each member of the population is assigned a biased probability of selection. Because of this increased probability of selection, the most fit members of a population are more likely to be selected for crossover than less fit members. However, there is always a possibility that a less fit member will be selected instead.

The crossover process continues until the size of the next generation is the same as the population size of the previous generation. After crossover has taken place, "mutation" is then applied to each member of the population. A typical mutation function is to assign a probability for flipping each bit. Thus, if a 10% value for mutation is assigned, then each bit of each member of the population has a 10% chance of being flipped (a 0 becomes a 1, or a 1 becomes a 0). After mutation is completed, each member of the new generation is evaluated for fitness and the process repeats for another generation. This process of evolving new populations continues until some criteria is met. The stopping criteria could be a) when the overall fitness of the population reaches a certain value, b) when the overall fitness

over several generations fails to change more than a specified value, or c) after a certain number of generations have passed.

There are many different ways to perform crossover and mutation. There are also many other "genetic operators" that have been used in GAs. Regardless of the details, the overall process remains the same: generate an initial population, evaluate the members of the population, generate a new population based upon the more fit members of the previous generation, and repeat the process until a certain stop criteria is achieved.

2.2.2 Applications of Genetic Algorithms

Genetic algorithms are very powerful search tools. By "search" it is meant that GAs are capable of pouring through a large number of potential solutions to find good solutions. Scheduling has been an area where genetic algorithms have proven very useful. The GA searches the space of potential schedules and finds those schedules which are most effective and maximize the desired criteria (such as minimizing idle time). For example, GAs are used by some airlines to schedule their flights (Dulay). An application of a GA to a financial problem (tactical asset allocation and international equity strategies) resulted in an 82% improvement in portfolio value over a passive benchmark model, and a 48% improvement over a non-GA model used to improve the passive benchmark (Dulay). GAs have also been applied to problems such as protein motif discovery through multiple sequence alignment and many other problems (Mendez, *et. al.)*.

For more information on genetic algorithms, can be found in David Goldberg's book "Genetic Algorithms in Search, Optimization, and Machine Learning." (Goldberg, 1989)

2.3 Hybrid Systems – Combining Neural Networks and Genetic Algorithms

There have been several systems developed that combine neural networks with genetic algorithms in various ways. These generally fall into three categories: (1) using a GA to determine the structure of a neural network, (2) using a GA to calculate the weights of a neural network, and (3) using a GA to both determine the structure and the weights of a neural network.

In the first category, a genetic algorithm is used in an attempt to find a good structure for a neural network. The process involves the GA evolving several structures and using the neural network as the fitness function to determine the fitness level of each structure. This technique can help eliminate the guess-work in deciding upon the structure of a neural network that can successfully be trained on the data.

In the second category, the genetic algorithm is used to evolve the weights of the network rather than using back propagation or some other technique for training connection weights. This can potentially result in quicker training of the network.

The third category uses the GA to generate not only the structure, but also the connection weights in the GA. For each structure in the population, another GA evolves weights. Using the evolved weights, the GA sets a fitness value for that network structure. Thus, two GAs are implemented with one running inside the

other. The neural network is the final result of both GAs, with one having determined the structure, and the other having determined the weights of the connections in the network.

Chapter 3

Statement of Problem

In a series of previous experiments involving artificial neural networks, a great deal of difficulty was encountered in trying to find a structure for the network which could quickly and accurately learn to identify four objects based upon the sounds they made. A previous set of experiments had already successfully classified the objects based upon image data. After several failed attempts where the structure for the network was "guessed" by the investigator, research into more reliable ways of obtaining network structures was performed.

One of the more intriguing possibilities was that of combining a neural network with a genetic algorithm. The genetic algorithm would create a population of potential structures for the neural network. The neural network would then briefly try each of the structures and report on the success of each. Using these values from the neural network, the genetic algorithm would then evolve a new population for the network to try. After several generations, a population of several "good" structures should result and could then be used to completely train the neural network.

The work that follows in this thesis shows the results of such an experiment. Using the same data from the successfully imaging experiments mentioned earlier as a benchmark, a new effort was undertaken to see if the concept of combining genetic algorithm with the neural network could produce results as good or better than the already successful network.

Chapter 4

Methodology

In order to use both image and sound data, four objects were selected that could produce sound. The objects were a toy duck which played music, a toy truck which made sounds like the engine was running, a race car toy which made sounds when it was moved, and a fire engine toy which made sounds when moved. For the rest of this thesis, the objects will be referred to as "Duck," "Truck," "Racecar," and "Fireman." Details into the methods employed and the reasons for choosing each are included in this chapter.

4.1 Setup

Setting up the experiment was an involved process. The four objects had to be imaged from many different angles, and the sounds from the objects had to be recorded. Then this data had to be converted from raw data into something that could be provided to the neural network as inputs.

4.1.1 Capturing the Images

In order to capture the images and sounds, a Logitech QuickCam was used. An "angle mat" was created by drawing long, intersecting lines on a sheet of paper. Each line was separated by five degrees. For each object, a center line was drawn on it in pencil so that each object could be set properly to any desired angle on the "angle mat." Images of all the objects were recorded every fifteen degrees. All images were

captured at the same time of day to ensure that lighting conditions were as controlled as possible.

4.1.2 Capturing Sounds

Using the microphone from the webcam, each sound from each object was recorded. The duck played three different songs, so each song was recorded. The racecar and the fireman each made multiple sounds, so each of those were recorded. The truck was the only object which made the same sound every time. More accurately, the duck would play music and quack with the song (for example, one of the songs was “Old MacDonald”). The truck sounded like the engine was running and it would periodically make a sound like the horn honking. The racecar and the fireman each made multiple different sounds. Care was taken in recording the sounds to ensure that there was as little ambient noise as possible.

4.1.3 Preparing the Data

For the image data, no real manipulation of the data needed to occur. However, the sound data required quite a bit of manipulation to be converted to useful data. Sounds were encoded in a .WAV format. The sounds were examined and trimmed in order to eliminate noise and empty space at the beginning and ends of the files. Because each of the objects made multiple sounds, with the exception of the duck, one sound from each of the objects was selected. Trying to train on multiple sounds for each object would have added a great deal of complexity to the problem. Then each file was broken into 10 pieces. Since each sound had a different length,

these pieces were not the same size. Each of the 10 pieces from the duck sound were the same size, but they were different sizes from the 10 pieces from the truck sound. In order for the sounds and images to be used in the neural network, the files had to be converted to raw data. To achieve this, a program was written that would take each byte of the file and normalize the value between zero and one before writing the resulting value out to a text file (a byte of 00000000 would equal zero, and a byte of 11111111 would equal one). In this manner, each byte of data was converted to a value between zero and one and placed within an easy-to-read text file. From there, it was easy to gather any number of desired inputs from any location in any file.

4.1.4 Gathering Inputs from Raw Data

Another program was written to gather data from the text files created using the processes described in section 4.1.3. The program asked for a number of bytes to be drawn from each file and used to create a training file for the neural network. These inputs were drawn from the center of the file. Thus, if 50 bytes were requested, the program would take the middle 50 bytes from each file and create a training set for the neural network, with each input file being used as a separate training example. Thus, if 50 byte sets were requested and there were 20 files, then 20 training examples of 50 bytes each would be created and merged into a single training set.

4.1.5 Training File Format

The training set was written as an ASCII text file with one training example per line and each value separated by a space. The last four values of each example would indicate which output should be turned on for that example. The outputs, in respective order, correlated to Duck, Fireman, Racecar, and Truck. The corresponding output value would be set to 1 and the others to 0. Thus, if a training example would result in being identified as the Duck, the values 1 0 0 0 would be appended to the end of the example. If an example corresponded to the Racecar, then the values 0 0 1 0 would be appended to the end of the example.

4.2 The Artificial Neural Network

The ANN was written in Visual Basic 6.0 (Microsoft), as Visual Basic provided a quick and easy way of building objects, as well as a graphical user interface that could be used to monitor the progress of the network and adjust values, if needed. Each neuron of the network was built as an object. The network was fully scalable, within the physical memory limitations of the PC. It supported any number of hidden layers, with any number of nodes in each layer. The training rate, maximum number of epochs, and error tolerance could all be modified from the Graphical User Interface (GUI). The current average error of learning is also displayed on the GUI and updated with each new epoch. The ANN program can be initiated in several ways. A network structure can be specified through the GUI and then a training set can be loaded. The program can also be passed several different structures via a text file. The latter method was implemented in order to allow the

neural network and the genetic algorithm to run concurrently without user-intervention.

The neural network has successfully learned several non-trivial classification problems. These include XOR and radar signature identification of aircraft, in which the network was trained to distinguish between two classes of aircraft based upon radar signals received at various angles. More importantly, the network had previously learned the image data for the four objects using a network of 300 inputs, 10 hidden nodes, and 4 outputs. Thus a known solution existed for the problem.

4.3 The Genetic Algorithm

The GA was implemented in C++ and was run under the Cygwin shell for Windows. The constraints on the GA were that each ANN structure could have a maximum of 5 layers: one input layer, three hidden layers, and one output layer. The input layer was fixed at 50 inputs and the output layer was fixed at 4 outputs. Each structure could have from zero to three hidden layers, with a maximum of 64 (2^6) nodes per layer. Each hidden layer was represented with 6 bits for a total of 18 bits per structure. Any layer which did not have at least one node in it was ignored. Thus, a bit stream resulting in values of 13 0 15 would be translated as only two hidden layers, with 13 nodes in the first layer and 15 nodes in the second layer. The algorithm used a population size of 100.

When evolving the population of possible solutions, the GA used two simple operators: crossover and mutation. The algorithm also used a “survival of the fittest” concept to preserve the most fit solutions from the previous generation. Thus, for

each new generation, one half of the population actually consisted of "surviving" members from the previous population. This was done by assigning a value to each member of the population in proportion to that member's contribution to the overall fitness of the population. More fit members were given larger values than less fit members. Then 50 members were drawn from the population based upon these weighted values. More fit members would have a higher likelihood of being moved to in the next generation. The rest of the next generation was obtained by combining the "survivors" that resulted from the selection just described. The theory behind this choice is that the survivors would be the most fit members of the previous generation and should result in reasonably fit "offspring." In order to obtain offspring, two of the survivors were selected at random and then the "crossover" operator was applied. The crossover operator randomly selected a position in the 18 bit sequence. Each bit from that position to the end of the sequence was then switched between the two sequences, potentially resulting in two new sequences. For example, given the two 18 bit sequences show in Figure 4.1, the crossover operator chooses bit 6 for the crossover. The results would be as shown:

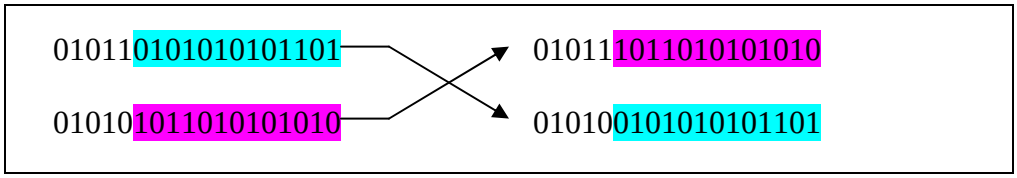

Figure 4.1. Example of results after the crossover operator is applied to two sequences.

Once the crossover operator had obtained two new offspring, each bit of the sequence would be tested for "mutation." Each bit was given a 10% chance of mutation. If a mutation occurred, the bit was flipped -- a 1 became a 0 and vice-versa.

This genetic algorithm implementation has been used successfully in a previous problem involving multiple sequence alignments of proteins. In the multiple sequence alignment problem, a series of proteins were examined to determine if they shared any similar subsequences. Similar subsequences can be exact matches, but typically contain amino acids that are commonly substituted for each other. Similarity scores were determined using the BLOSUM62 substitution matrix. Alignment of the sequences involves inserting "gaps" into sequences so that the similar subsequences are aligned with each other.

4.4 Combining the Neural Network and the Genetic Algorithm

in order to eliminate the need for constant user interaction in the process and thus improve overall performance and speed, it was necessary that both the NN and GA programs run concurrently. This was implemented through a file-passing protocol.

The neural network would wait until a file called "ga.done" was created. If the file was not present, then the neural network would look again in 30 seconds. Once the "ga.done" file was found, it was deleted and the file "ga.txt" was loaded. The "ga.txt" file contained the path to the training set to be used by the neural network, the number of epochs to train each network in the file, the number of networks contained in the file, and the structure of each network. The neural network would then create a copy of the file as "ga.#.txt" where # was the current generation in the genetic algorithm. Thus, "ga.1.txt" would be the structures output from the

first generation of the genetic algorithm. In this manner, each generation of structures could be saved for future reference. The neural network would then attempt to train using each structure in the file and the specified training set for the number of epochs specified in the file. After the last epoch, the ANN would write the fitness result (calculated as 1 – error) of the network to a file called “nn.txt” and then attempt the next structure in the file. After the last structure had been attempted, the ANN copied the “nn.txt” file to “nn.#.txt” for archival purposes as described earlier. Then the neural network would write a file called “nn.txt.go” and then go back into wait mode.

Upon start up, the genetic algorithm created an initial population of structures and then wrote out the “ga.txt” file. When the file was completed, the file “ga.done” was written, which activated the neural network and placed the GA into wait mode. The GA would continuously look for the file “nn.txt.go” which indicated the ANN had finished evaluating the structures. Once the file was found, the GA deleted it and loaded the file “nn.txt” which contained the fitness values for each member of the population. Using these fitness values, the GA would then generate a new population based upon the procedures described in earlier sections. The resulting population would be written to “ga.txt” and then the file “ga.done” would be created. This would again activate the neural network and place the GA into wait mode. This process would continue until a desired number of generations had been produced by the genetic algorithm. The number of generations was passed as a command-line argument to the genetic algorithm.

4.5 Final Remarks on Methodology

Although difficulties in learning the sound data were what led to this experiment, the image data was chosen because a neural network had successfully learned the data. The goal of this experiment was to validate the concept before using it to find a structure for learning the sound data. The previously successful network only contained a single hidden layer with only 10 nodes. This would indicate that the number of inputs provided enough data to nearly stand alone, and would not be of sufficient difficulty to prove the concept of combining the GA with the ANN. If a guess at a trivial neural network topology was sufficient to successfully learn the data, then the problem would need to be more complicated in order to truly test if the GA would show improvement over random guesses. To make the problem more difficult, the number of inputs was reduced from 300 to 50. By using fewer inputs, less information would be provided to the network, thus making successful classification a more difficult task. Furthermore, by restricting the number of inputs, the amount of time required to run the ANN for all 100 structures in each generation was significantly reduced.

Chapter 5

Results

5.1 Crossover and Mutation, 5 epochs

In the first experiment, each ANN structure was trained for only 5 epochs. From the results, the average fitness seemed to peak around generation 40. For each generation, the best fit member of the population remained the same, with the exception of generation 94, where a sudden spike occurred and then vanished in the next generation. This spike had a fitness score of 0.427, while the best fit member of each other generation only had a 0.378 fitness score. It is also worth noting that this popular member vanished at generation 13, but promptly reappeared in the following generation. The average fitness did not show much improvement after the first 25 generations or so. Figure 5.1 shows the most fit member at each generation, and the average fitness of each generation.

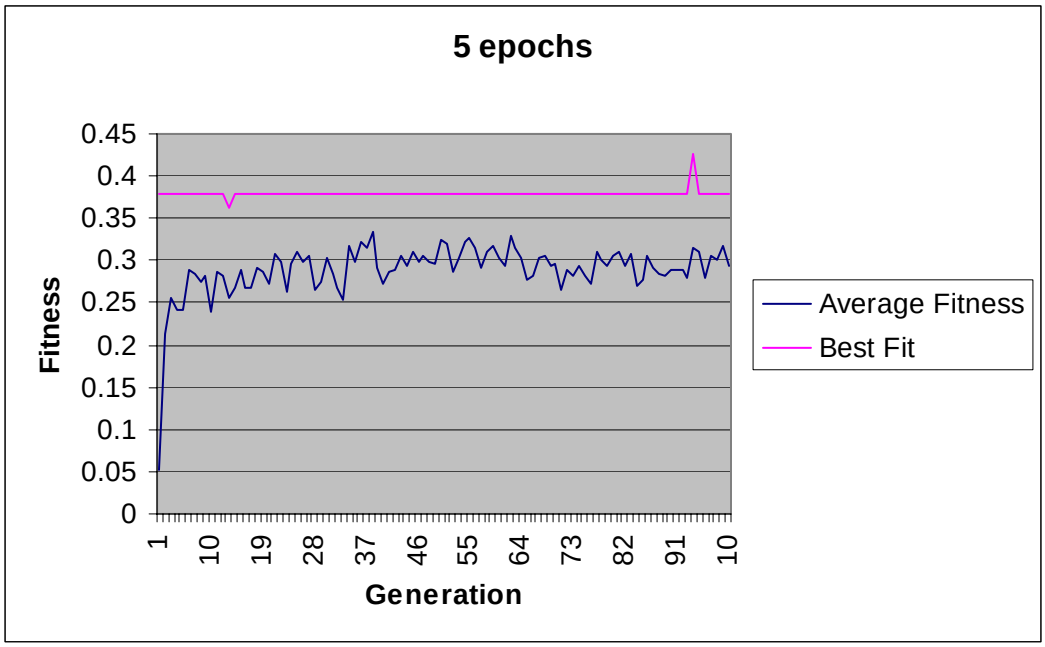

Figure 5.1. Results of evaluating each topology after 5 epochs.

The well-conserved "best fit" member of the population, as well as the "spike," were both run through the neural network to see if they could successfully learn the training data. Both structures failed. They would continue to show improvement up to a certain limit, at which point both of the structures ceased to show any improvement in learning, regardless of the number of epochs they were allowed to run. The spike had a structure of 50-44-23-12-4, which was 50 inputs, 44 neurons in the first hidden layer, 23 neurons in the second hidden layer, 12 neurons in the third hidden layer, and 4 output neurons. The better conserved member had a structure of 50-56-49-13-4. Both networks trained to an error of approximately 0.75, which is unacceptable (75% error rate).

5.2 Crossover and Mutation, 20 epochs

For the second experiment, each structure was trained longer in the neural network in order to evaluate its fitness. Rather than the 5 epochs used in the first experiment, each structure was trained for 20 epochs before a fitness value was given. This time, only 22 generations were run. This was done to save time, and because observations from the prior experiment showed little change after generation 20. This is further justified by the graph of the average fitness. Figure 5.2 shows very little change after the first few generations as the average hovers around the 0.30 mark for each generation following the first one. There appears to be no real benefit from letting the algorithm continue to run.

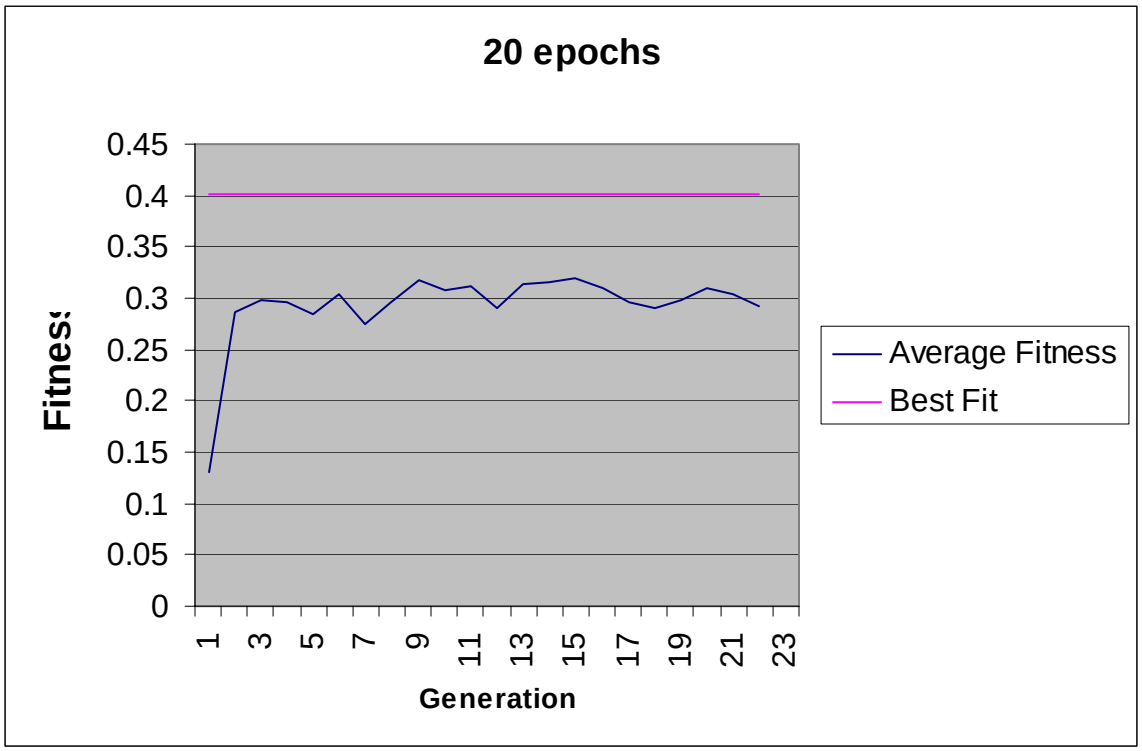

Figure 5.2. Results of evaluating each topology after 20 epochs.

As was observed in the first experiment, the best fit member from the first generation is well conserved through each generation. However, unlike the first experiment, there was no improvement upon the best fit member after the first generation.

The best fit member from this experiment had a fitness value of 0.402 and a structure of 50-44-26-17-4. This structure was run through the neural network and also failed to learn the data regardless of the number of epochs it was allowed to train. As in the first experiment, the network obtained an error of 0.75 and then failed to improve.

5.3 Mutation only, 20 epochs

After obtaining the results of the first two experiments, further research was done to see what results have been reported in similar studies. A paper by Garcia-Pedrajas *et. al.* (2004) suggested that the standard crossover operator when used in conjunction with a neural network is actually just a mutation. This is because of the permutations possible in how the network is encoded within the genetic algorithm. Taking note of this, and the fact that in both of the previous experiments the best fit member from the first generation was carried through each generation, a third experiment was conducted that used only the mutation operator, without crossover. When creating the next generation, it was populated entirely using the weighted probability method mentioned previously. Thus, the next generation would be populated with the more fit members from the previous generation. The mutation operator was then applied to each member of the population. The result from this

method, in theory, should be a population of fit members which have been tweaked slightly from generation to generation. As in the previous experiment, each structure was trained for 20 epochs in the neural network before its fitness value was determined.

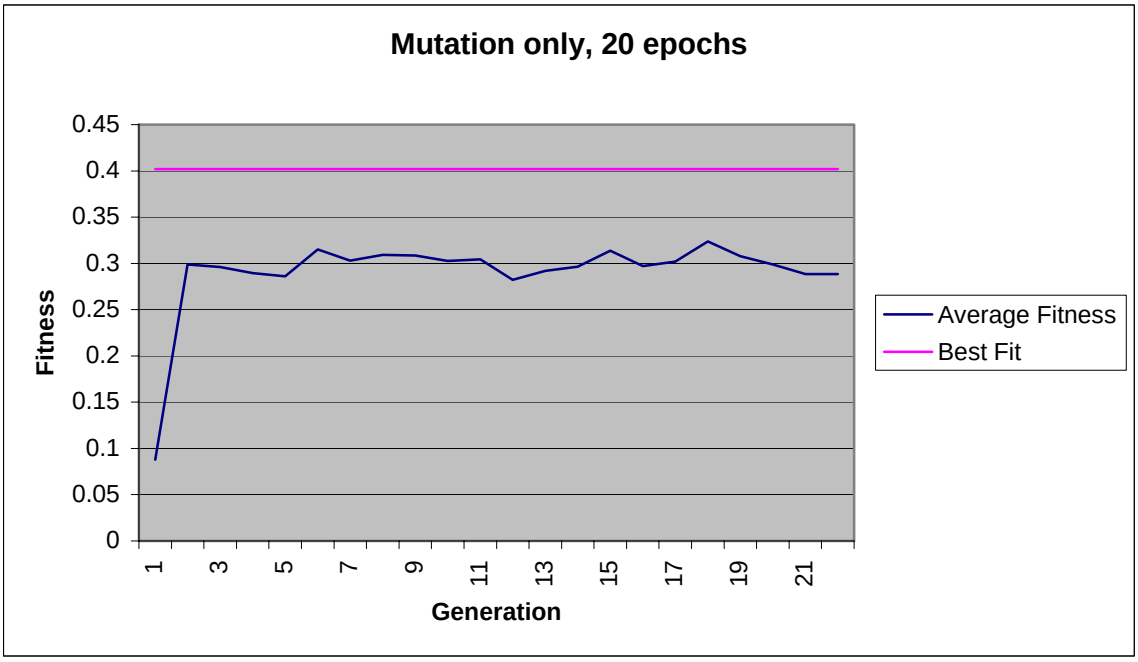

Figure 5.3. Results of topologies evolved using mutation only, evaluated after 20 epochs.

Just as in the previous experiment, the best fit member of the first generation was preserved through the entire algorithm, despite the mutation factor. In fact, the frequency with which it occurred in each generation grew dramatically, reaching 27% by the last generation. However, this applies only to the value of the best fit member and not the actual structure itself. Several similar structures resulted in the same fitness value. Examination of these structures showed an overwhelming preference for either 16 or 17 neurons in the final hidden layer. This feature was present in 68%

of the final population. It is also worth noting that the best fit structure from the previous experiment had the same fitness value as the best fit member from this experiment, 0.402, and had 17 neurons in its last hidden layer.

5.4 Discussion of Results

In each experiment, one of the population members obtained by a random selection in the first generation was maintained throughout each generation of the algorithm. In the first experiment, where only 5 epochs were used for evaluation, there were two generations which were exceptions. In generation 13, the best fit member vanished but then promptly returned in the following generation. In generation 94, a better fit member was found, but it vanished in the next generation, despite the weighted probability of it being selected to remain.

Table 5.1 summarizes the best fit members obtained from each experiment, including the "spike" obtained in the first experiment.

Experiment	*Best Fitness Value*	*Structure of Network*
W/selection, 5 epochs	0.378	50-56-49-13-4
(spike at generation 94)	0.427	50-44-23-12-4
W/selection, 20 epochs	0.402	50-44-26-17-4
Mutation only, 20 epochs	0.402	50-53-17-4

Table 5.1. Topologies with the highest fitness values from each experiment.

Figure 5.4 shows the average fitness of each generation from all three experiments. The graph only includes the first 22 generations of the first experiment.

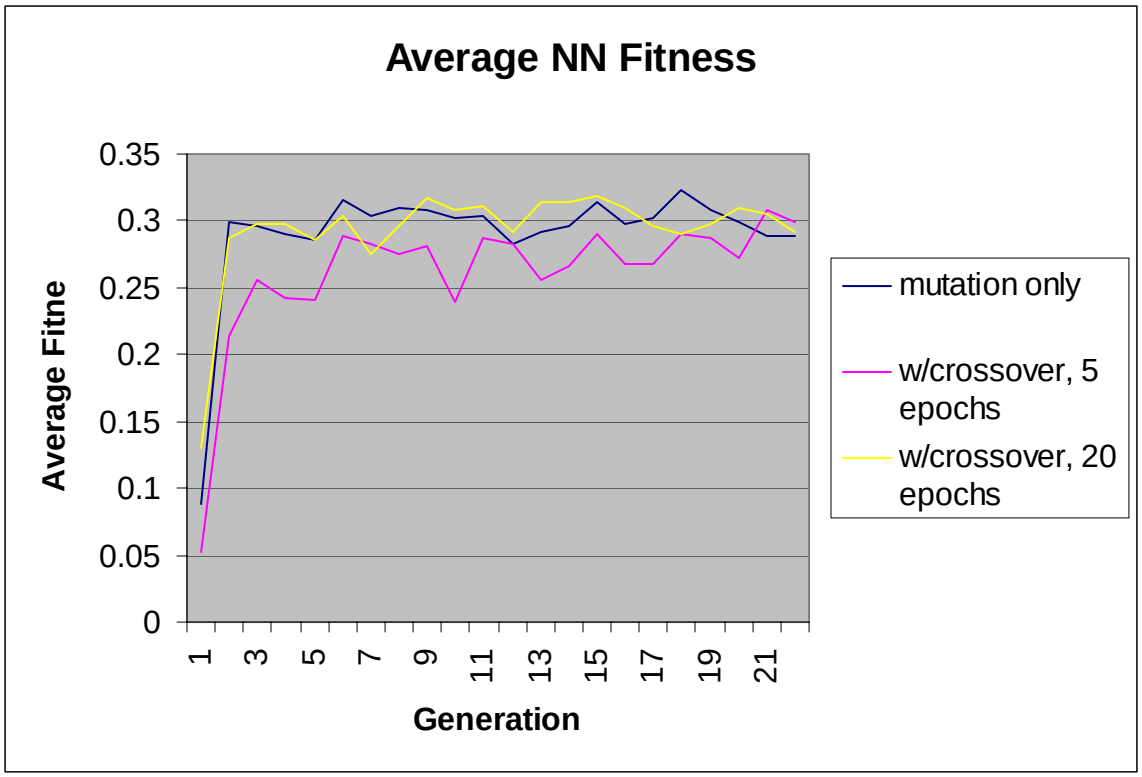

Figure 5.4. Comparison of the average population fitness from the three experiments.

From Figure 5.4, it appears that the experiment using only 5 epochs failed to produce results comparable to the other two experiments. However, from Figure 5.1, one can see that the results of the experiment do reach the same levels as the other two after about 10 generations. This seems intuitive. After all, the longer a network is trained, the more accurate the results should be from the network, thus resulting in higher fitness values. Thus from the first experiment, one can observe that the genetic algorithm is indeed improving the overall fitness of the population. However, it failed to result in a better overall structure, and took significantly more time to run than the other two experiments. The second and third experiments, where the networks were trained longer before given a fitness score, show that the genetic

algorithm appears to provide very little, if any, improvement. The curves in Figure 5.4 are fairly level for both of these experiments, indicating that all of the improvements occurred in the first few generations. In none of the three experiments did the genetic algorithm appear to improve the best structure, which was the entire purpose of these experiments.

The structures produced from the three experiments, as shown in Table 5.1, provide some interesting results. First of all, in each of the structures, the number of nodes in the last hidden layer is between 12 and 17 neurons. When combined with the results from experiment 3, where 68% of the final population had 16 or 17 neurons in the last hidden layer, this seems to indicate that perhaps there is some ideal ratio between the number of output nodes and the number of neurons in the preceding layer. It is also interesting to note that each of the structures also has a "wedge" shape in that from the first hidden layer to the output layer the number of neurons in each layer decreases.

It is also worth noting that the best structure from the third experiment only had two hidden layers, while the best structures from the other experiments all had three hidden layers. This is especially interesting in that the fitness score for this network structure was the same as the best structure from the second experiment which had three layers. Both structure have 17 nodes in the last hidden layer. This would seem to indicate that the number of nodes in the last hidden layer is more important than the number of layers in the structure.

To further compare the results of these structure, three structures “guessed” at were also trained in the neural network to see how they compared to the structures resulting from the previous three experiments. None of the “guess” structures were successful in learning the data, either. The three structures were 50-10-4, 50-30-4, and 50-15-15-4.

Chapter 6

Conclusion

6.1 Summary

In each of the three experiments, the best fit structure from the first generation was carried consistently throughout following generations. This indicates that the genetic algorithm did not improve upon the best "guess" from the first generation. The genetic algorithm implementation used for these experiments had previously been used to solve a non-trivial problem, thus making it unlikely that the results observed were due to a problem with the GA implementation. Furthermore, the experiment using only mutation yielded results equivalent to the two experiments which included crossover. While the genetic algorithm improved the overall fitness of the population, it failed to improve upon the best fit member, which came from the initial random population of the first generation.

6.2 Contributions

After the surprising results of these experiments, further research into similar experiments was conducted to see what results had been obtained by other researchers. Fiszelew *et al.* (ND) compared neural network structures resulting from this hybrid method to the results of random neural network structures. The error rate after training showed only a 2% difference, indicating little improvement with the hybrid method. However, when the two were compared in classification accuracy, the hybrid structure was 10% more accurate in classifications, i.e., 85% accuracy

compared with 75% accuracy from the randomly selected structure. In a paper by Optiz and Shavlik (1997), genetic algorithms were used to find neural network topologies. The results showed up to a 33% improvement in error rates using the genetic algorithm when compared to other optimization methods. However, in the experiment showing the most significant improvement in test-set error, the standard neural network had an error of 7.83% while the network topology derived from the genetic algorithm had an error of 4.08%. Comparatively, this is a significant improvement, but in terms of actual difference the genetic algorithm method reduced the error in the test-set by less than 4%. Furthermore, they reported that the process took 4 CPU days to finish. With a difference of less than 4% resulting after 4 CPU days, the gain in performance does not justify the time required to achieve the improvement.

From the findings reported by Fiszelew *et al.*, Optiz and Shavlik, and the experiments reported in this thesis, the benefit of employing a genetic algorithm to obtain an improved neural network topology is arguable. The only observable benefit would be in the accuracy of the network when classifying previously unseen data. While the 10% improvement reported by Fiszelew *et al.* is nice, the benefit was still less than half of the inaccuracy rate, and showed only a 2% improvement in learning the training set. Optiz and Shavlik showed only a 4% difference between the error rates of the generated topology and the standard one, requiring 4 CPU days for the GA process to run.

In conclusion, it would appear that using a genetic algorithm to obtain a neural network topology has little benefit when compared to the amount of time required to implement such a system and then allow the system to run and evolve a topology. However, it is possible that the results obtained in these experiments were skewed because too few input neurons were used. The benefit in training is miniscule, and the benefit in accuracy could be merely coincidental. One could save considerable time and effort by generating several random topologies and testing each for a few epochs before selecting one to go forward with. Several different topologies are capable of learning the same data, and even the same topology can learn the data differently if random starting weights, or a different learning rate, are assigned. Thus, there can be several potentially successful topologies for any given data set.

6.3 Limitations

Limiting the number of inputs to 50 appears to have made the problem too complicated to be solved with a neural network. Unfortunately, this means that none of the topologies were able to learn the data, and so no comparisons in accuracy could be made. Furthermore, the constraints placed upon the network topologies might have been too restrictive. It is possible, but unlikely, that a topology with more layers, or more nodes in each layer, might have been able to successfully learn the training set. The constraints on number of inputs were chosen because a simple topology was successful with 300 inputs. The other constraints on topology were chosen so that the evaluations performed by the neural network would proceed faster.

Observations regarding the resulting topologies from the genetic algorithm showed that almost all of the topologies used all three hidden layers. There were very few topologies with only two hidden layers, and none that had only one layer. This seems to indicate that genetic algorithm needed another operator, perhaps another form of mutation, that would remove a layer from a given topology. This would provide more variation in the population and expand the search space significantly.

Another possible limitation could be that all four of the objects used for training contained the color yellow in significant amounts. If the inputs drawn from the images corresponded to the yellow areas of each object, this could make training on the images nearly impossible.

6.4 Future Work

A follow up study should take careful precautions to ensure that the data being learned by the neural networks can, in fact, by learned. It should begin by allowing a large number of inputs, evolving networks, training them, and then comparing the accuracy of each. Then the number of inputs should be reduced and followed with another round. In this manner, the accuracy of random topologies versus that of the evolved topologies can be compared and analyzed with fewer and fewer inputs. While this process would be very time consuming, it should be able to definitively answer the question as to whether or not evolving a topology is worth the amount of time required to do so. In order to perform a truly exhaustive comparison, this should be repeated with data sets from different domains.

References

Dale, N., Weems, C., and Headington, M. (2002) *Programming and Problem Solving with C++*. Third Edition. Jones and Bartlett.

Dulay, Narankar. *Genetic Algorithms.* Surprise 96 Journal on-line.
http://www.doc.ic.ac.uk/~nd/surprise_96/journal/vol4/tcw2/report.html#Application
Downloaded on April 24, 2005 at 5:21pm.

Fiszelew, A., Britos, P., Perichisky, G., and Garcia-Martinez, R *Automatic Generation of Neural Networks based on Genetic Algorithms.* http://www.presidentekennedy.br/resi/edicao02/artigo01.pdf. Downloaded on March 6, 2005 at 9:23 pm.

GAITS: Global Analytic Information Technology Services. *Biometrics Overview: GAITS Portal Control System (PCS).* GAITS corporate web site
http://www.gaits.com/biometrics_face.asp. Downloaded on April 24, 2005 at 4:05pm.

Garcia-Pedrajas, N., Ortiz-Boyer, D., Hervas-Martinez, C. (2004) *An alternative approach for neural network evolution with a genetic algorithm: Crossover by combinatorial optimization.* University of Granda, Soft Computing and Intelligent Information Systems research group. http://sci2s.ugr.es/keel/publicaciones/uco/articulos/sax2005.pdf. Downloaded on March 6, 2005 at 10:45pm.

Goldberg, D. *Genetic Algorithms in Search, Optimization, and Machine Learning.* (1989) Addison-Wesley.

Grobstein, Paul. (1994) *Variability in Brain Function and Behavior.* In V.S. Ramachandran (Ed.), The Encyclopedia of Human Behavior, vol. 4. pp. 447-458.
http://serendip.brynmawr.edu/bb/EncyHumBehav.html
Downloaded on October 6, 2003 at 6:16pm.

Guan, Ying-Hua. *Reexamining the modality effect from the perspective of Baddeley's working memory model.*
http://www.iwm-kmrc.de/workshops/visualization/guan.pdf
Downloaded on October 9, 2003 at 5:09pm.

Guyon, I., Bromley, J., Mati, N., Schenkel, M., and Weissman, H. *Penactive: A Neural Net System for Recognizing On-Line Handwriting.*
http://sherry.ifi.unizh.ch/513647.html
Downloaded on May 25, 2005 at 5:54pm.

Hosom, John-Paul, et. al. (1999) *Training Neural Networks for Speech Recognition.* Center for Spoken Language Understanding. Oregon Graduate Institute of Science and Technology.
http://cslu.cse.ogi.edu/tutordemos/nnet_training/tutorial.html
Downloaded on October 12, 2003 at 12:15pm.

Kosko, Bart. (1996) *Neural Networks and Fuzzy Systems.* Prentice Hall.

Lyon, Richard F. and Yaeger, Larry S. (1996) *On-Line Hand-Printing Recognition with Neural Networks.* Fifth International Conference on Microelectronics for Neural Networks and Fuzzy Systems. IEEE Computer Society Press.

Mendez, J., Falcon, A., Lorenzo, J. *A Procedure for Biological Sensitive Pattern Matching in Protein Sequences.* Intelligent Systems Institute. University Las Palmas de Gran Canaria, Spain.
http://www.iusiani.ulpgc.es/images/publicaciones/matching.pdf
Downloaded on August 30, 2004 at 12:00pm.

Mount, David W. (2004) *Bioinformatics: Sequence and Genome Analysis.* Second Edition. Cold Spring Harbor Laboratory Press.

Optiz, D. W. and Shavlik, Jude W. (1997) *Connectionist Theory Refinement: Genetically Searching the Space of Network Topologies.* Journal of Artificial Intelligence Research, vol 6, pp. 177-209.

Pal, Sankar K. and Mitra, Sushmita. (1992) *Multilayer Perceptron, Fuzzy Sets, and Classification.* IEEE Transactions on Neural Networks, vol. 3, no. 5, pp. 683-697.